EMBEDDED LINUX CONFERENCE

@

THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA

V4L2 M2M as the driver framework for Video Processing IP

Karthik Poduval /Amazon Lab126

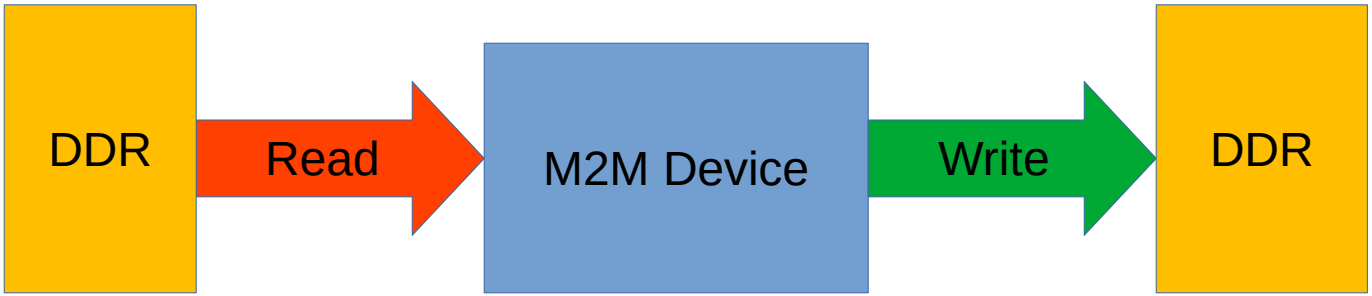
#ossummit

THE LINUX FOUNDATION

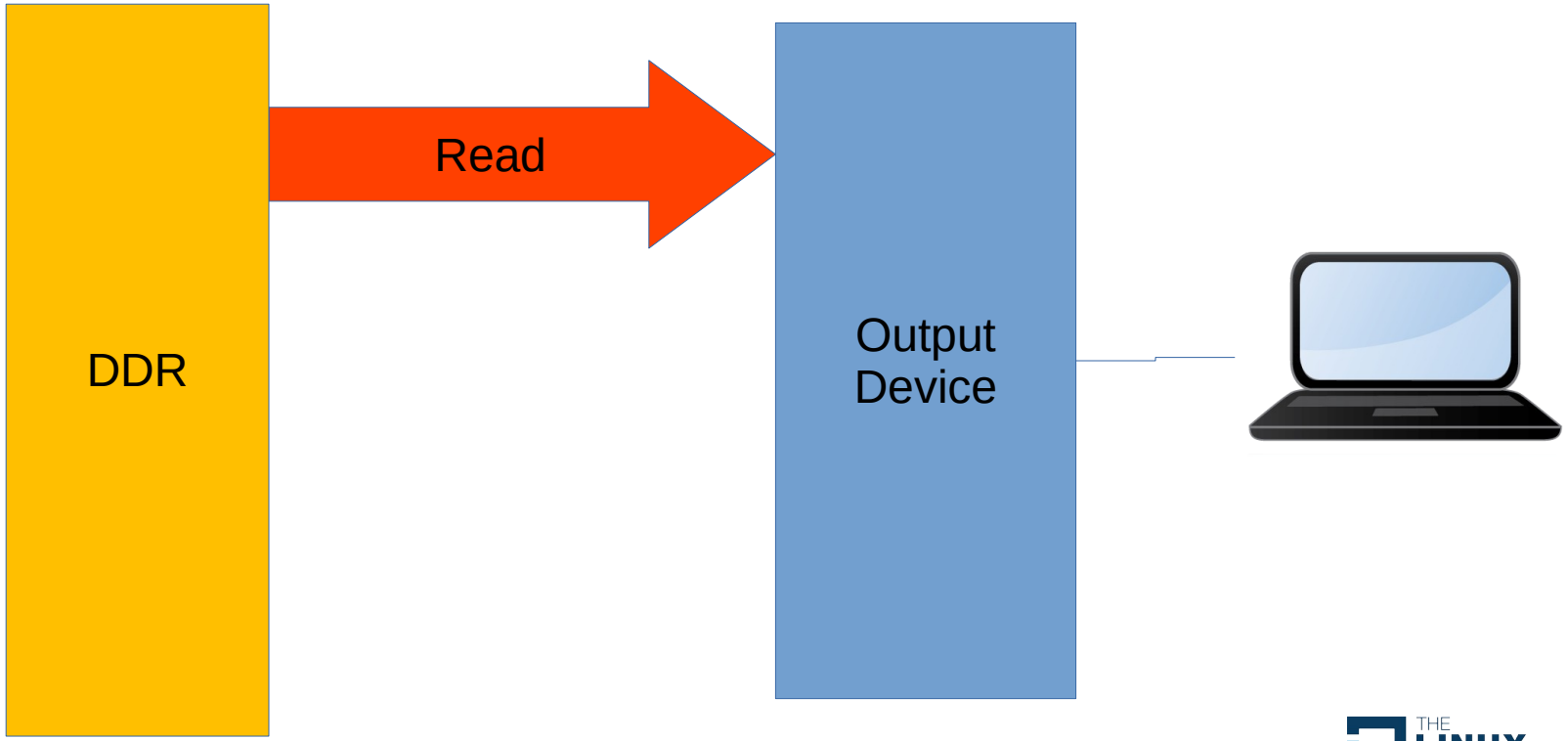
What is V4L2 M2M ?

- A driver framework for memory to memory devices
- Memory to memory devices take data from memory do some processing and write out processed data back to memory
- Different from traditional V4L2 devices which are either
  - Capture
  - Output
- V4L2 M2M will have both output and capture
- V4L2 M2M supports multiple contexts which traditional V4L2 devices do not (usually)

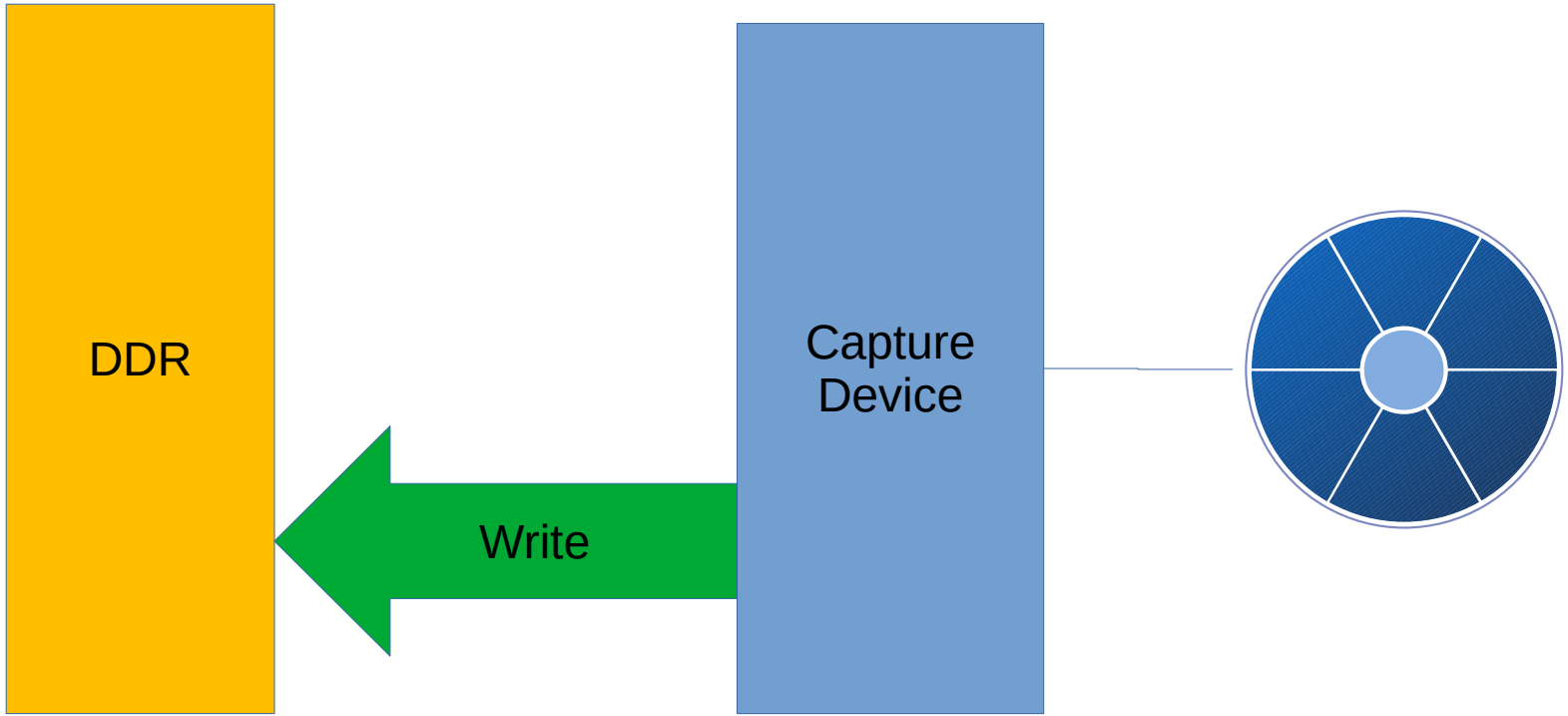
Memory to Memory Device Logical View



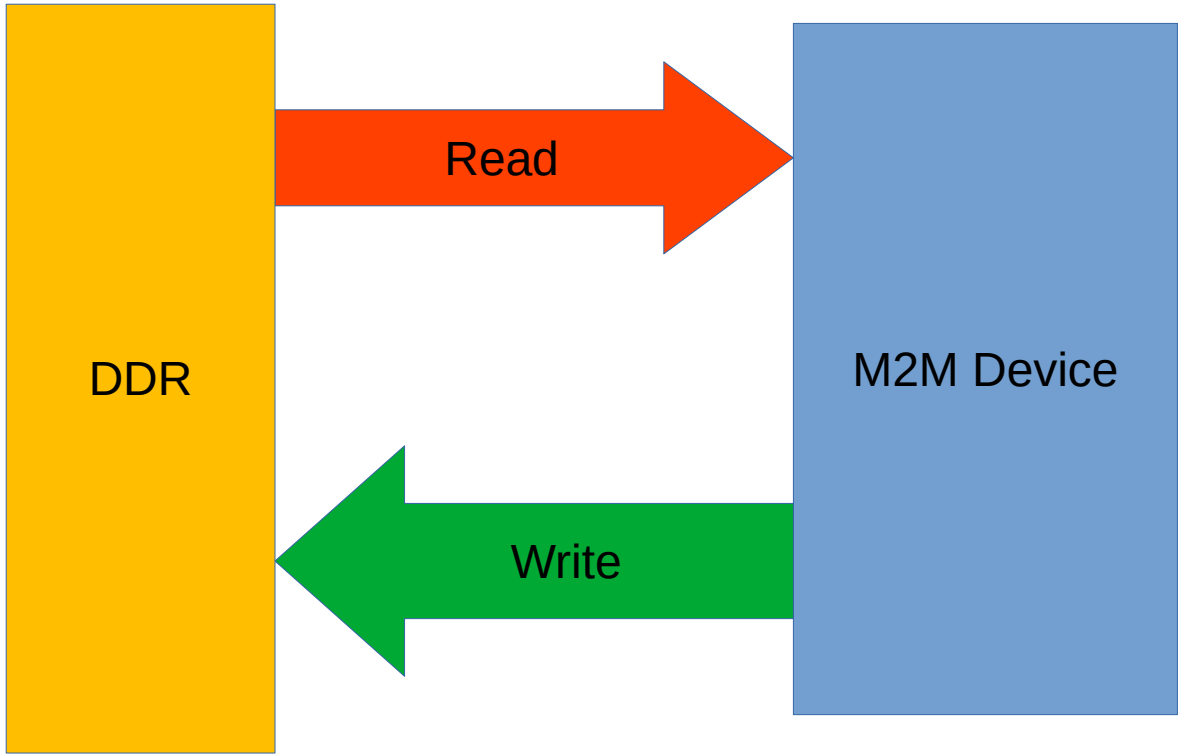
V4L2 Output Device



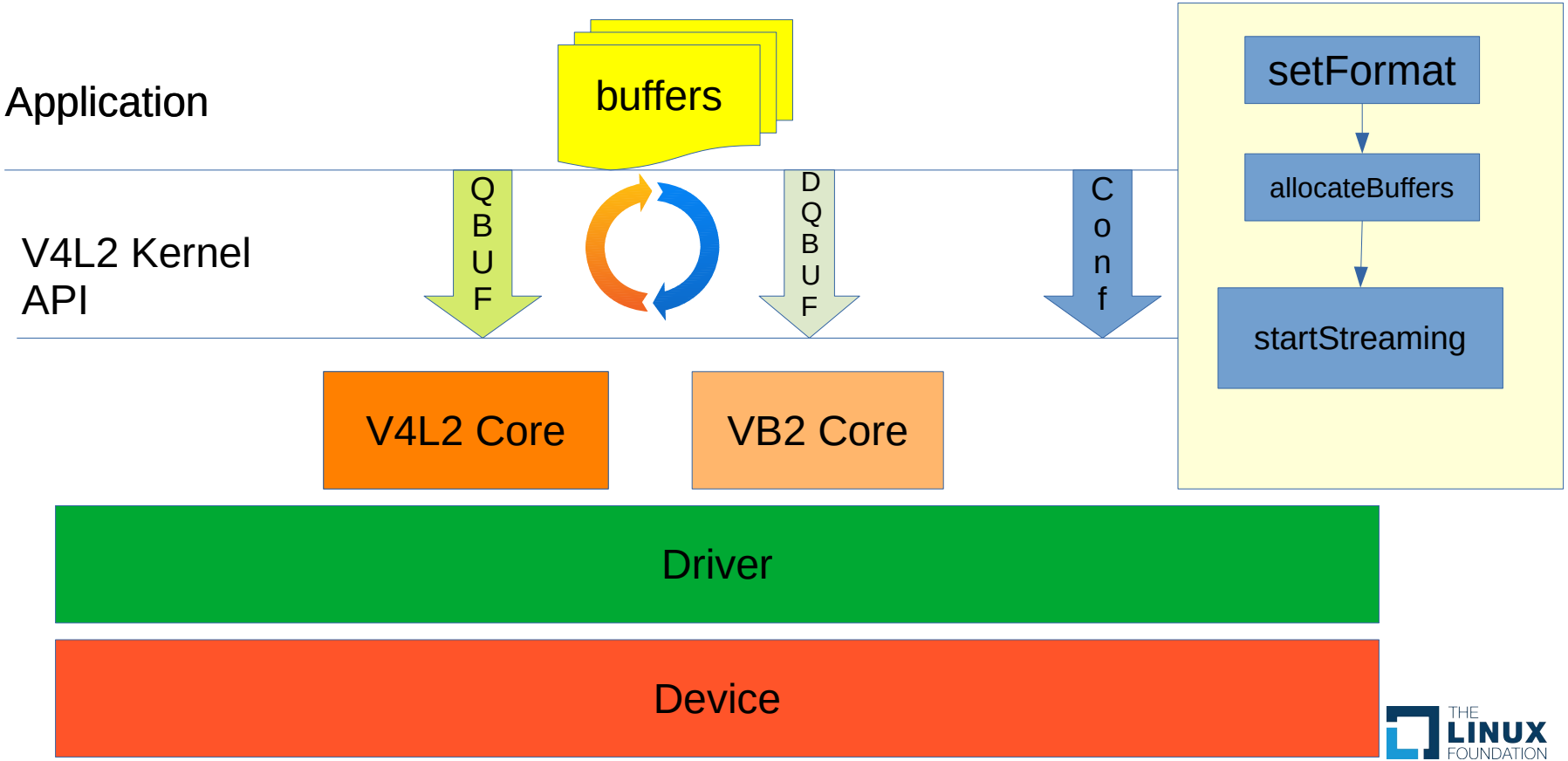
V4L2 Capture Device



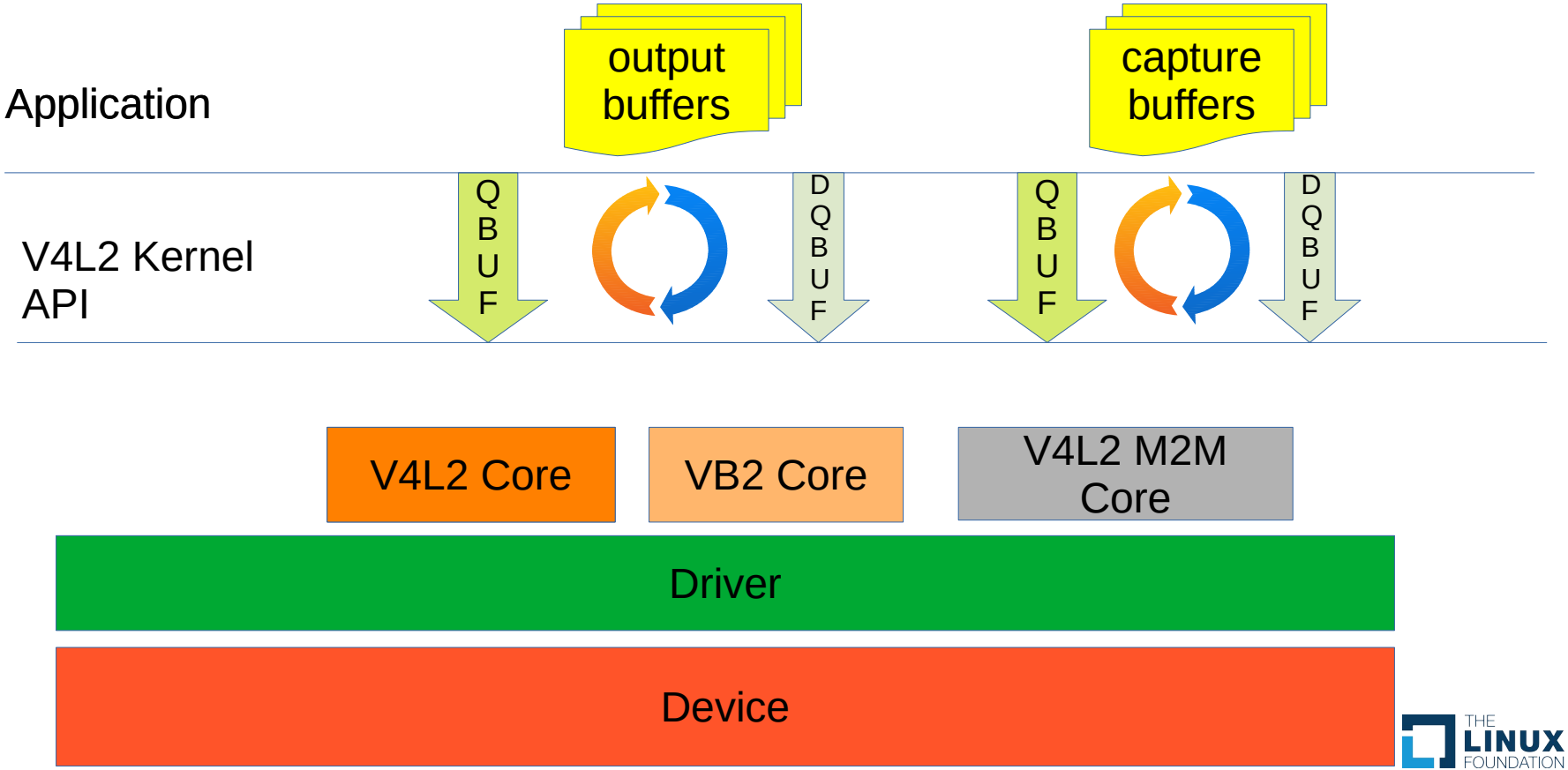
V4L2 M2M Device



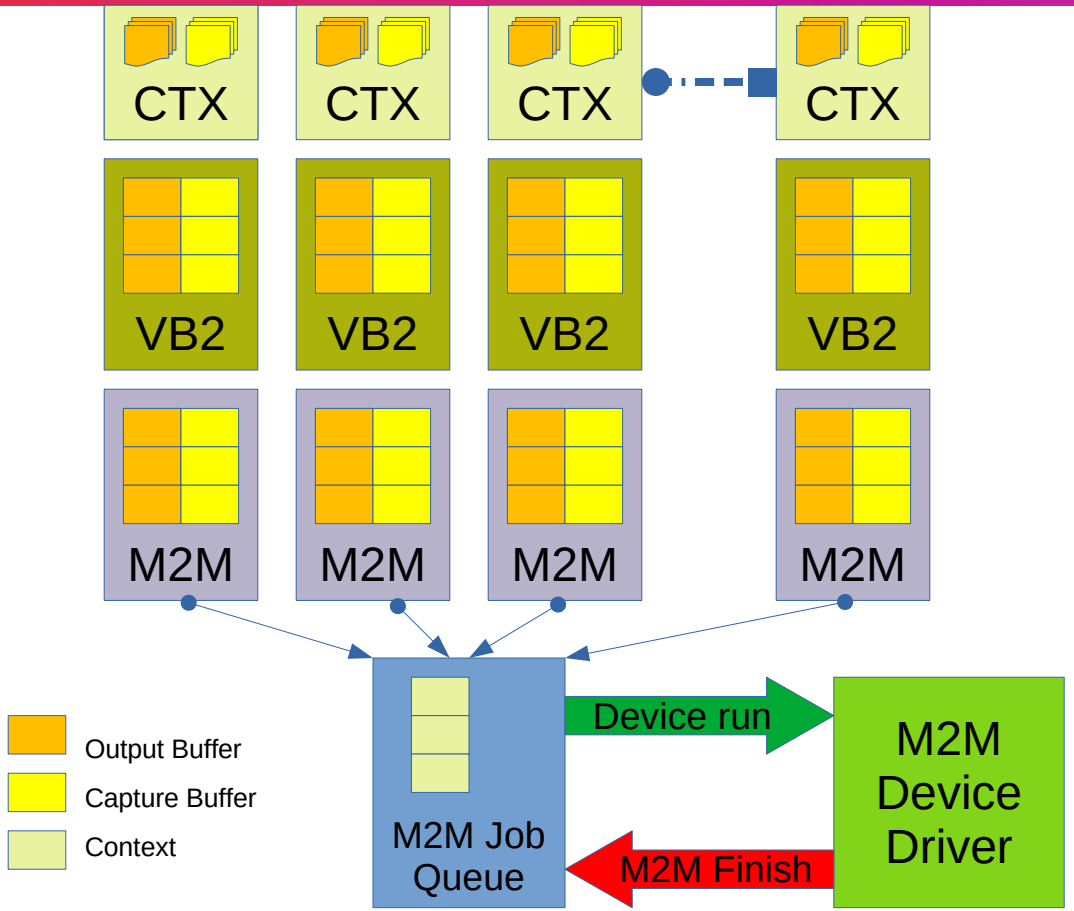
Typical V4L2 Application Workflow



V4L2 M2M Application Workflow



V4L2 M2M Architecture



- Individual contexts queue buffers on their output and capture VB2 queues
- From there it makes it to M2M per context ready queue when VB2 call driver's queue_buffer callback
- Once required number of output and capture buffers are ready, move context to job queue
- The device_run callback of driver in invoked
- Device run pulls necessary number of buffers from output and capture ready queue for the current context
- Once device finishes processing the job, call vb2_buffer done to return buffer to application and m2m_job_finish to remove context from job queue.

An Example M2M Scaler Device

- A virtual M2M scaler device
- Device is built using
  STB Image Resize Library
- Device is virtual and emulated using QEMU
- Using yocto and Linux kernel 5.15 we build a device driver for the virtual device
- Using libcamera C++ Library we also demonstrate a simple downscaling example

M2M Scaler Device Datasheet

Virtual memory-2-memory Scaler

This is the data sheet for a QEMU based Virtual memory-2-memory scaler. The scaler can perform upscaling as well as downscaling. No ratios need to be supplied instead the actual input and output sizes are supplied. The scaler only supports BGR image format and both input and output must be in this format.

Register Map

Offset 0x00 - Input Configuration 1

Bits	Description
0:15	Input Width
16:31	Input Height

Offset 0x04 - Input Configuration 2

Bits	Description
0:15	Input Stride (in bytes)
16:31	Reserved

Offset 0x08 – Output Configuration 1

Bits	Description
0:15	Output Width
16:31	Output Height

Offset 0x0C - Output Configuration 2

Bits	Description
0:15	Output Stride (in bytes)
16:31	Reserved

Offset 0x10 – Input Buffer DMA Address

Bits	Description
0:31	Input Buffer DMA Address

Offset 0x14 – Output Buffer DMA Address

Bits	Description
0:31	Output Buffer DMA Address

Offset 0x18 – Control and Status

Bits	Description
0	Start processing
1	Enable Interrupts
2	Reset
3:4	Status 0: idle 1: processing 2: done 3: done but has error

Programming Model

1. **Program input width and height register 0x0**
2. **Program input stride register 0x4**
3. **Program output width and height register 0x8**
4. **Program output stride register 0xc**
5. **Program input buffer DMA address register 0x10**
6. **Program output buffer DMA address register 0x14**
7. **Write the start bit (bit 0) of register 0x18**
8. **Either poll status or wait for interrupt (if interrupt enabled 0x14:1)**
9. **Discard output frame if error status bits**

Driver – Platform Driver

```
static const struct of_device_id m2m_scaler_dt_ids[] = {
        { .compatible = "virtual,m2m-scaler", .data = NULL },
        { },
};
MODULE_DEVICE_TABLE(of, m2m_scaler_dt_ids);

static struct platform_driver m2m_scaler_driver = {
        .probe          = m2m_scaler_probe,
        .remove         = m2m_scaler_remove,
        .driver         = {
                .name   = MEM2MEM_NAME,
                .of_match_table = m2m_scaler_dt_ids,
        },
};

module_platform_driver(m2m_scaler_driver);
```

```
    m2m-scaler@9010000 {
        interrupt-names = "irq";
        interrupts = <0x0 0x2 0x1>;
        reg = <0x0 0x9010000 0x0 0x1000>;
        compatible = "virtual,m2m-scaler";
    };
```

Driver – probe

```
static int m2m_scaler_probe(struct platform_device *pdev)
{
        struct m2m_scaler *device;
        struct device *dev = &pdev->dev;
        struct resource *res;
        struct video_device *vfd;
        struct v4l2_device *v4l2_dev = &device->v4l2_dev;
        int irq;
        int ret = 0;

        device = devm_kzalloc(dev, sizeof(*device), GFP_KERNEL);
        if(!device)
                return -ENOMEM;

        res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
        device->mmio = devm_ioremap_resource(dev, res);
        if (IS_ERR(device->mmio))
                return PTR_ERR(device->mmio);

        device->regmap = devm_regmap_init_mmio(dev, device->mmio, &m2m_scaler_regmap_config);
        if(IS_ERR(device->regmap)) {
                dev_err(dev, "regmap init failed\n");
                return PTR_ERR(device->regmap);
        }
        if(m2m_scaler_regfield_alloc(dev, device)) {
                dev_err(dev, "reg field alloc failed\n");
                return -ENODEV;
        }

        irq = platform_get_irq(pdev, 0);
        if (irq < 0)
                return irq;

        ret = devm_request_threaded_irq(dev, irq, NULL, m2m_scaler_irq_handler,
                        IRQF_ONESHOT, dev_name(dev), device);
        if (ret < 0) {
                dev_err(dev, "Failed to request irq: %d\n", ret);
                return ret;
        }
```

- Use regmap to program device
- Allocate regfields to make programming easier
- Register threaded IRQ so we can return completed frame from IRQ itself

Driver – probe

```
ret = v4l2_device_register(&pdev->dev, &device->v4l2_dev);
if (ret) {
        dev_err(dev, "could not register video device rc=%d\n", ret);
        return ret;
}

device->video_dev = m2m_scaler_video_dev;
vfd = &device->video_dev;
vfd->lock = &device->lock;
vfd->v4l2_dev = &device->v4l2_dev;

/* set the video device private data structure to struct m2m_scaler
 * instance */
video_set_drvdata(vfd, device);

/* also set the platform private to the same */
platform_set_drvdata(pdev, device);

snprintf(vfd->name, sizeof(vfd->name), "%s", MEM2MEM_NAME);

device->m2m_dev = v4l2_m2m_init(&m2m_ops);
if (IS_ERR(device->m2m_dev)) {
        v4l2_err(v4l2_dev, "Failed to init mem2mem device\n");
        ret = PTR_ERR(device->m2m_dev);
        goto err_v4l2;
}

ret = video_register_device(vfd, VFL_TYPE_VIDEO, 0);
if (ret) {
        v4l2_err(v4l2_dev, "Failed to register video device\n");
        goto err_m2m;
}

regmap_field_write(device->enable_interrupts, 1);
```

- Register v4l2 device
- Initialize m2m_ops
- Register video device
- Enable interrupts

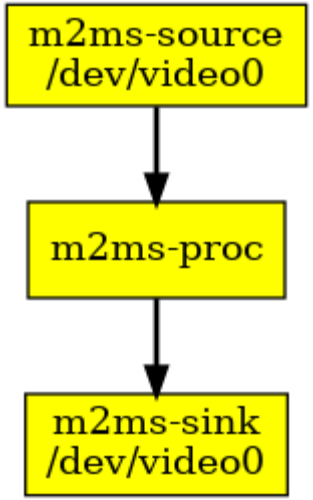
Driver – probe

```
#ifdef CONFIG_MEDIA_CONTROLLER
        device->mdev.dev = &pdev->dev;
        strscpy(device->mdev.model, MEM2MEM_NAME, sizeof(device->mdev.model));
        strscpy(device->mdev.bus_info, "platform:m2m-scaler", sizeof(device->mdev.bus_info));
        media_device_init(&device->mdev);
        device->mdev.ops = &m2m_media_ops;
        device->v4l2_dev.mdev = &device->mdev;

        ret = v4l2_m2m_register_media_controller(device->m2m_dev, vfd, MEDIA_ENT_F_PROC_VIDEO_COMPOSER);
        if(ret) {
                v4l2_err(v4l2_dev, "Failed to init media controller\n");
                goto err_m2m;
        }

        ret = media_device_register(&device->mdev);
        if(ret) {
                v4l2_err(v4l2_dev, "Failed to register media device\n");
                goto err_m2m;
        }
#endif

        return 0;
```



- Media controller is also supported
- Mostly for purposes media request API on output video node

Driver – ops

```
static const struct v4l2_file_operations m2m_scaler_fops = {
        .owner          = THIS_MODULE,
        .open           = m2m_scaler_open,
        .release        = m2m_scaler_release,
        .poll           = v4l2_m2m_fop_poll,
        .unlocked_ioctl = video_ioctl2,
        .mmap           = v4l2_m2m_fop_mmap,
};

static const struct video_device m2m_scaler_video_dev = {
        .name           = MEM2MEM_NAME,
        .vfl_dir        = VFL_DIR_M2M,
        .fops           = &m2m_scaler_fops,
        .device_caps    = V4L2_CAP_VIDEO_M2M | V4L2_CAP_STREAMING,
        .ioctl_ops      = &m2m_scaler_ioctl_ops,
        .minor          = -1,
        .release        = video_device_release_empty,
};

static const struct v4l2_m2m_ops m2m_ops = {
        .device_run     = m2m_scaler_device_run,
};

#ifdef CONFIG_MEDIA_CONTROLLER
static struct media_device_ops m2m_media_ops = {
        .req_validate = vb2_request_validate,
        .req_queue = v4l2_m2m_request_queue,
};
#endif
```

- m2m_media_ops: to support media request API
- m2m_ops: device_run callback actually runs the m2m job on the device
- m2m_scaler_fops: file operations to create and release context

Driver – open

```
static int m2m_scaler_open(struct file *file)
{
        struct m2m_scaler *device = video_drvdata(file);
        struct m2m_scaler_ctx *ctx = NULL;
        struct v4l2_device *v4l2_dev = &device->v4l2_dev;
        int rc = 0;

        if (mutex_lock_interruptible(&device->lock))
                return -ERESTARTSYS;
        ctx = kzalloc(sizeof(*ctx), GFP_KERNEL);
        if (!ctx) {
                rc = -ENOMEM;
                goto open_unlock;
        }

        v4l2_fh_init(&ctx->fh, video_devdata(file));
        file->private_data = &ctx->fh;
        ctx->device = device;

        ctx->fh.m2m_ctx = v4l2_m2m_ctx_init(device->m2m_dev, ctx, &queue_init);

        if (IS_ERR(ctx->fh.m2m_ctx)) {
                rc = PTR_ERR(ctx->fh.m2m_ctx);

                v4l2_fh_exit(&ctx->fh);
                kfree(ctx);
                goto open_unlock;
        }

        v4l2_fh_add(&ctx->fh);
```

- Create a context
- Pass queue_init callback
- Initialize driver context related data structure

Driver – open

```
	/* set default format */
	ctx->fmt[FMT_OUTPUT].type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
	ctx->fmt[FMT_OUTPUT].fmt.pix.pixelformat = FORMAT;
	ctx->fmt[FMT_OUTPUT].fmt.pix.width = DEFAULT_WIDTH;
	ctx->fmt[FMT_OUTPUT].fmt.pix.height = DEFAULT_HEIGHT;
	ctx->fmt[FMT_CAPTURE].type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
	ctx->fmt[FMT_CAPTURE].fmt.pix.pixelformat = FORMAT;
	ctx->fmt[FMT_CAPTURE].fmt.pix.width = DEFAULT_WIDTH;
	ctx->fmt[FMT_CAPTURE].fmt.pix.height = DEFAULT_HEIGHT;
```

```
struct m2m_scaler_ctx {
	struct v4l2_fh		fh;
	struct m2m_scaler	*device;
	struct v4l2_format	fmt[2];
	uint64_t		sequence;
};
```

- Driver context initialization involves setting up the default output and capture formats
- For this driver its fixed to V4L2_PIX_FMT_RGB24

Driver – release

```
static int m2m_scaler_release(struct file *file)
{
        struct m2m_scaler *device = video_drvdata(file);
        struct m2m_scaler_ctx *ctx = file2ctx(file);
        struct v4l2_device *v4l2_dev = &device->v4l2_dev;

        v4l2_dbg(1, debug, v4l2_dev, "Releasing instance %p\n", ctx);

        v4l2_fh_del(&ctx->fh);
        v4l2_fh_exit(&ctx->fh);
        mutex_lock(&device->lock);
        v4l2_m2m_ctx_release(ctx->fh.m2m_ctx);
        mutex_unlock(&device->lock);
        kfree(ctx);

        return 0;
}
```

- Uninitialize and free up the context

Driver – queue_init

```
static int queue_init(void *priv, struct vb2_queue *src_vq,
                      struct vb2_queue *dst_vq)
{
        struct m2m_scaler_ctx *ctx = priv;
        int ret;

        src_vq->type = V4L2_BUF_TYPE_VIDEO_OUTPUT;
        src_vq->io_modes = VB2_MMAP | VB2_DMABUF;
        src_vq->drv_priv = ctx;
        src_vq->buf_struct_size = sizeof(struct v4l2_m2m_buffer);
        src_vq->ops = &m2m_scaler_qops;
        src_vq->mem_ops = &vb2_dma_contig_memops;
        src_vq->timestamp_flags = V4L2_BUF_FLAG_TIMESTAMP_COPY;
        src_vq->lock = &ctx->device->lock;
        src_vq->dev = ctx->device->v4l2_dev.dev;

        ret = vb2_queue_init(src_vq);
        if (ret)
                return ret;

        dst_vq->type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
        dst_vq->io_modes = VB2_MMAP | VB2_DMABUF;
        dst_vq->drv_priv = ctx;
        dst_vq->buf_struct_size = sizeof(struct v4l2_m2m_buffer);
        dst_vq->ops = &m2m_scaler_qops;
        dst_vq->mem_ops = &vb2_dma_contig_memops;
        dst_vq->timestamp_flags = V4L2_BUF_FLAG_TIMESTAMP_COPY;
        dst_vq->lock = &ctx->device->lock;
        dst_vq->dev = ctx->device->v4l2_dev.dev;

        return vb2_queue_init(dst_vq);
}
```

- Setup output and capture video node VB2 ops
- For this device only physically contiguous memories are supported, hence use vb2_dma_contig_memops

```
static const struct vb2_ops m2m_scaler_qops = {
        .queue_setup     = m2m_scaler_queue_setup,
        .buf_prepare     = m2m_scaler_buf_prepare,
        .buf_queue       = m2m_scaler_buf_queue,
        .start_streaming = m2m_scaler_start_streaming,
        .stop_streaming  = m2m_scaler_stop_streaming,
        .wait_prepare    = vb2_ops_wait_prepare,
        .wait_finish     = vb2_ops_wait_finish,
};
```

Driver – queue_setup

```
static int m2m_scaler_queue_setup(struct vb2_queue *vq,
                        unsigned int *nbuffers, unsigned int *nplanes,
                        unsigned int sizes[], struct device *alloc_devs[])
{
        struct m2m_scaler_ctx *ctx = vb2_get_drv_priv(vq);
        struct m2m_scaler *device = ctx->device;
        struct v4l2_device *v4l2_dev = &device->v4l2_dev;
        unsigned int count = *nbuffers;

        struct v4l2_format *fmt;

        fmt = m2m_scaler_get_format(ctx, vq->type);
        if(IS_ERR(fmt))
                return -EINVAL;

        *nplanes = 1;
        sizes[0] = fmt->fmt.pix.sizeimage;

        v4l2_dbg(1, debug, v4l2_dev, "get %d buffer(s) of size %d each.\n", count, sizes[0]);

        return 0;
}
```

- This callback tells VB2 the actual number of planes and size of each plane

Driver – buf_prepare

```
static int m2m_scaler_buf_prepare(struct vb2_buffer *vb)
{
        struct m2m_scaler_ctx *ctx = vb2_get_drv_priv(vb->vb2_queue);
        struct m2m_scaler *device = ctx->device;
        struct v4l2_device *v4l2_dev = &device->v4l2_dev;

        struct v4l2_format *fmt;

        fmt = m2m_scaler_get_format(ctx, vb->type);
        if(IS_ERR(fmt))
                return -EINVAL;


        v4l2_dbg(1, debug, v4l2_dev, "type: %d\n", vb->vb2_queue->type);

        vb2_set_plane_payload(vb, 0, fmt->fmt.pix.sizeimage);

        return 0;
}
```

- This callback is used to prepare the buffer before queuing it to its VB2 queue
- In this call the plane payload size is set

Driver – queue, start and stop streaming

```
static void m2m_scaler_buf_queue(struct vb2_buffer *vb)
{
        struct vb2_v4l2_buffer *vbuf = to_vb2_v4l2_buffer(vb);
        struct m2m_scaler_ctx *ctx = vb2_get_drv_priv(vb->vb2_queue);

        v4l2_m2m_buf_queue(ctx->fh.m2m_ctx, vbuf);
}

static int m2m_scaler_start_streaming(struct vb2_queue *q, unsigned int count)
{
        struct m2m_scaler_ctx *ctx = vb2_get_drv_priv(q);

        ctx->sequence = 0;
        return 0;
}

static void m2m_scaler_stop_streaming(struct vb2_queue *q)
{
        struct m2m_scaler_ctx *ctx = vb2_get_drv_priv(q);
        struct vb2_v4l2_buffer *vbuf;

        for (;;) {
                if (V4L2_TYPE_IS_OUTPUT(q->type))
                        vbuf = v4l2_m2m_src_buf_remove(ctx->fh.m2m_ctx);
                else
                        vbuf = v4l2_m2m_dst_buf_remove(ctx->fh.m2m_ctx);
                if (vbuf == NULL)
                        return;
                v4l2_m2m_buf_done(vbuf, VB2_BUF_STATE_ERROR);
        }
}
```

- Queue calls to v4l2_m2m_buf_queue
- Start streaming initialized sequence number
- Stop streaming drains the output and capture queues and returns the buffers as errors

Driver – ioctl ops

```
static const struct v4l2_ioctl_ops m2m_scaler_ioctl_ops = {
        .vidioc_querycap        = m2m_scaler_querycap,

        .vidioc_enum_fmt_vid_cap = m2m_scaler_enum_fmt,
        .vidioc_g_fmt_vid_cap   = m2m_scaler_g_fmt,
        .vidioc_try_fmt_vid_cap = m2m_scaler_try_fmt,
        .vidioc_s_fmt_vid_cap   = m2m_scaler_s_fmt,

        .vidioc_enum_fmt_vid_out = m2m_scaler_enum_fmt,
        .vidioc_g_fmt_vid_out   = m2m_scaler_g_fmt,
        .vidioc_try_fmt_vid_out = m2m_scaler_try_fmt,
        .vidioc_s_fmt_vid_out   = m2m_scaler_s_fmt,

        .vidioc_reqbufs         = v4l2_m2m_ioctl_reqbufs,
        .vidioc_querybuf        = v4l2_m2m_ioctl_querybuf,
        .vidioc_qbuf            = v4l2_m2m_ioctl_qbuf,
        .vidioc_dqbuf           = v4l2_m2m_ioctl_dqbuf,
        .vidioc_prepare_buf     = v4l2_m2m_ioctl_prepare_buf,
        .vidioc_create_bufs     = v4l2_m2m_ioctl_create_bufs,
        .vidioc_expbuf          = v4l2_m2m_ioctl_expbuf,

        .vidioc_streamon        = v4l2_m2m_ioctl_streamon,
        .vidioc_streamoff       = v4l2_m2m_ioctl_streamoff,
};
```

- Only format ioctl are implemented
- Since both output and capture nodes support same format, common set of callbacks are used

Driver – ioctl ops

```c
static int m2m_scaler_try_fmt(struct file *file, void*priv, struct v4l2_format *f)
{
        if(f->fmt.pix.pixelformat != FORMAT)
                f->fmt.pix.pixelformat = FORMAT;

        if(f->fmt.pix.width > MAX_WIDTH)
                f->fmt.pix.width = MAX_WIDTH;

        if(f->fmt.pix.height > MAX_HEIGHT)
                f->fmt.pix.width = MAX_HEIGHT;

        f->fmt.pix.sizeimage = f->fmt.pix.width * f->fmt.pix.height * 3; //For the supported RGB format
        return 0;
}
static int m2m_scaler_enum_fmt(struct file *file, void *priv,
                               struct v4l2_fmtdesc *f)
{
        struct m2m_scaler_ctx *ctx = (struct m2m_scaler_ctx *) priv;
        struct v4l2_format *fmt;

        fmt = m2m_scaler_get_format(ctx, f->type);

        if(IS_ERR(fmt))
                return -EINVAL;

        /* only one format supported */
        if(f->index > 1)
                return -EINVAL;

        f->pixelformat = FORMAT;

        return 0;
}
```

```c
static int m2m_scaler_g_fmt(struct file *file, void *priv,
                                struct v4l2_format *f)
{
        struct m2m_scaler_ctx *ctx = (struct m2m_scaler_ctx *) priv;
        struct v4l2_format *fmt;

        fmt = m2m_scaler_get_format(ctx, f->type);

        if(IS_ERR(fmt))
                return -EINVAL;

        f->fmt.pix = fmt->fmt.pix;

        return 0;
}

static int m2m_scaler_s_fmt(struct file *file, void *priv,
                                struct v4l2_format *f)
{
        struct m2m_scaler_ctx *ctx = (struct m2m_scaler_ctx *) priv;
        struct v4l2_format *fmt;

        m2m_scaler_try_fmt(file, priv, f);

        fmt = m2m_scaler_get_format(ctx, f->type);

        if(IS_ERR(fmt))
                return -EINVAL;

        fmt->fmt.pix = f->fmt.pix;

        return 0;
}
```

- Ensure resolution limits are met

Driver – device run

```
static void m2m_scaler_device_run(void *priv)
{
        struct m2m_scaler_ctx *ctx = priv;
        struct m2m_scaler *device = ctx->device;
        struct vb2_v4l2_buffer *src_buf, *dst_buf;
        dma_addr_t input_addr, output_addr;
        uint16_t iwidth, iheight, istride;
        uint16_t owidth, oheight, ostride;
        struct v4l2_device *v4l2_dev = &device->v4l2_dev;

        src_buf = v4l2_m2m_next_src_buf(ctx->fh.m2m_ctx);
        dst_buf = v4l2_m2m_next_dst_buf(ctx->fh.m2m_ctx);

        /* program the scaler*/

        /* reset the m2m scaler HW */
        regmap_field_write(device->reset, 1);

        /* program resolution info */
        iwidth = ctx->fmt[FMT_OUTPUT].fmt.pix.width;
        iheight = ctx->fmt[FMT_OUTPUT].fmt.pix.height;
        istride = iwidth*3;
        regmap_field_write(device->input_width, iwidth);
        regmap_field_write(device->input_height, iheight);
        regmap_field_write(device->input_stride, istride);

        owidth = ctx->fmt[FMT_CAPTURE].fmt.pix.width;
        oheight = ctx->fmt[FMT_CAPTURE].fmt.pix.height;
        ostride = owidth*3;
        regmap_field_write(device->output_width, owidth);
        regmap_field_write(device->output_height, oheight);
        regmap_field_write(device->output_stride, ostride);

        v4l2_dbg(1, debug, v4l2_dev, "%s:iw=%d ih=%d is=%d\n", __func__, iwidth, iheight, istride);
        v4l2_dbg(1, debug, v4l2_dev, "%s:ow=%d oh=%d os=%d\n", __func__, owidth, oheight, ostride);

        /* program dma addresses */
        input_addr = vb2_dma_contig_plane_dma_addr(&src_buf->vb2_buf, 0);
        output_addr = vb2_dma_contig_plane_dma_addr(&dst_buf->vb2_buf, 0);
        regmap_field_write(device->input_addr, input_addr);
        regmap_field_write(device->output_addr, output_addr);

        /* start processing */
        regmap_field_write(device->start_processing, 1);
}
```

- Here is where the actual programming to the device occurs
- Once a job is given to the device, V4L2 M2M framework will wait for the job to complete before calling the device_run again
- Source and destination buffers are kept on their respective queues
- device_run need not be synchronous, which is typically the case
- Device starts the m2m processing and delivers an interrupt after processing is done

Driver – job complete

```
static irqreturn_t m2m_scaler_irq_handler(int irq, void *dev_id)
{
        struct m2m_scaler *device = (struct m2m_scaler *)dev_id;
        struct m2m_scaler_ctx *curr_ctx;
        struct vb2_v4l2_buffer *src_vb, *dst_vb;
        uint32_t status;
        int vb2_status;

        curr_ctx = (struct m2m_scaler_ctx*) v4l2_m2m_get_curr_priv(device->m2m_dev);
        if(curr_ctx) {
                regmap_field_read(device->status, &status);
                switch(status) {
                        case STATUS_DONE:
                                vb2_status = VB2_BUF_STATE_DONE;
                                break;

                        case STATUS_DONE_ERROR:
                        default:
                                vb2_status = VB2_BUF_STATE_ERROR;

                }


                /* return the src and dst buffers back to V4L2 M2M layer to return to application */
                src_vb = v4l2_m2m_src_buf_remove(curr_ctx->fh.m2m_ctx);
                dst_vb = v4l2_m2m_dst_buf_remove(curr_ctx->fh.m2m_ctx);
                src_vb->sequence = dst_vb->sequence = curr_ctx->sequence++;
                v4l2_m2m_buf_done(src_vb, vb2_status);
                v4l2_m2m_buf_done(dst_vb, vb2_status);
                v4l2_m2m_job_finish(device->m2m_dev, curr_ctx->fh.m2m_ctx);

        }

        return IRQ_HANDLED;
}
```

- Upon job completion, device will issue an interrupt
- In the ISR check to see if the m2m processioning job was successful
- Now remove the source and destination buffers from their m2m queues
- Return buffers back to their VB2 queues
- Signal m2m framework about job completion

Test Application

```
class M2MScaler {
public:
        M2MScaler(Size input, Size output);
        int run();

private:
        std::unique_ptr<DeviceEnumerator> enumerator_;
        std::shared_ptr<MediaDevice> media_;
        V4L2M2MDevice *m2mScaler_;

        std::vector<std::unique_ptr<FrameBuffer>> captureBuffers_;
        std::vector<std::unique_ptr<FrameBuffer>> outputBuffers_;

        unsigned int outputFrames_;
        unsigned int captureFrames_;
        Size inputSize_;
        Size outputSize_;
        std::vector<MappedBuffer> inputMappedBuffer_;
        std::vector<MappedBuffer> outputMappedBuffer_;
};
```

```
int main()
{
        Size input(640, 480);
        Size output(320, 240);
        M2MScaler *s = new M2MScaler(input, output);

        assert(s->run() == 0);
}
```

- Simple dowscaling test
- Input buffer is 640x480 while output buffer is expected to be 320x240

Test Application

```
using namespace std;
using namespace libcamera;


class M2MScaler {
public:
        M2MScaler(Size input, Size output) : captureFrames_(0), outputFrames_(0) {
                inputSize_ = input;
                outputSize_ = output;

                enumerator_ = DeviceEnumerator::create();
                if (!enumerator_) {
                        cerr << "Failed to create device enumerator" << endl;
                        assert(1);
                }

                if (enumerator_->enumerate()) {
                        cerr << "Failed to enumerate media devices" << endl;
                        assert(1);
                }

                DeviceMatch dm("m2ms");
                dm.add("m2ms-source");
                dm.add("m2ms-sink");

                media_ = enumerator_->search(dm);
                if (!media_) {
                        cerr << "No m2ms device found" << endl;
                        assert(1);
                }


        }
```

- create M2MScaler class that uses libcamera's helper classes
- Use the device enumerator that uses media controller API to discover the device driver
- Take input and output buffer sizes as class argument (Size is a libcamera class that stores width and height)

Test Application

```
int run() {
        constexpr unsigned int bufferCount = 1;

        EventDispatcher *dispatcher = Thread::current()->eventDispatcher();
        int ret;

        MediaEntity *entity = media_->getEntityByName("m2ms-source");
        m2mScaler_ = new V4L2M2MDevice(entity->deviceNode());
        if (m2mScaler_->open()) {
                cerr << "Failed to open M2M Scaler device" << endl;
                return -1;
        }

        V4L2VideoDevice *capture = m2mScaler_->capture();
        V4L2VideoDevice *output = m2mScaler_->output();

        V4L2DeviceFormat format = {};
        if (capture->getFormat(&format)) {
                cerr << "Failed to get capture format" << endl;
                return -1;
        }

        format.size = outputSize_;

        if (capture->setFormat(&format)) {
                cerr << "Failed to set capture format" << endl;
                return -1;
        }

        format.size = inputSize_;

        if (output->setFormat(&format)) {
                cerr << "Failed to set output format" << endl;
                return -1;
        }
```

- Find and open the video node (this will create an m2m context)
- Find the default output and capture format
- Update output and capture resolution (our driver only supports one format but different resolutions within)

Test Application

```
capture->bufferReady.connect(this, &M2MScaler::receiveCaptureBuffer);
output->bufferReady.connect(this, &M2MScaler::outputBufferComplete);

int cookie = 0;
for (const std::unique_ptr<FrameBuffer> &buffer : captureBuffers_) {
        if (capture->queueBuffer(buffer.get())) {
                std::cout << "Failed to queue capture buffer" << std::endl;
        }
        buffer->setCookie(cookie++);
        /* mmap the buffer */
        MappedFrameBuffer map(buffer.get(), MappedFrameBuffer::MapFlag::ReadWrite);
        assert(map.isValid());
        outputMappedBuffer_.push_back(std::move(map));
}

for (const std::unique_ptr<FrameBuffer> &buffer : outputBuffers_) {
        if (output->queueBuffer(buffer.get())) {
                std::cout << "Failed to queue output buffer" << std::endl;
        }
        /* mmap the buffer */
        MappedFrameBuffer map(buffer.get(), MappedFrameBuffer::MapFlag::ReadWrite);
        assert(map.isValid());
        assert(map.planes()[0].size() == bbb_splash_rgb_len);
        memcpy(map.planes()[0].data(), bbb_splash_rgb, map.planes()[0].size());
        inputMappedBuffer_.push_back(std::move(map));
}

ret = capture->streamOn();
if (ret) {
        cerr << "Failed to streamOn capture" << endl;
}

ret = output->streamOn();
if (ret) {
        cerr << "Failed to streamOn output" << endl;
}
```

- Install callbacks to handle output and capture buffers when they get returned from the driver
- Queue the output and capture buffers
- Memory map the buffers as well
- memcpy know buffer (640x480 resolution) to mapped output buffers
- Stream on both output and capture video nodes

Test Application

```
	Timer timeout;
	timeout.start(50000);
	while (timeout.isRunning()) {
		dispatcher->processEvents();
		if (captureFrames_ > 4)
			break;
	}

	cerr << "Output " << outputFrames_ << " frames" << std::endl;
	cerr << "Captured " << captureFrames_ << " frames" << std::endl;

	if (captureFrames_ < 4) {
		cerr << "Failed to capture 30 frames within timeout." << std::endl;
		return -1;
	}

	ret = capture->streamOff();
	if (ret) {
		cerr << "Failed to StreamOff the capture device." << std::endl;
		return -1;
	}

	ret = output->streamOff();
	if (ret) {
		cerr << "Failed to StreamOff the output device." << std::endl;
		return -1;
	}

	return 0;
```

- After stream on wait for required number off output and capture buffers to cycle through the driver
- Once 4 capture buffers cycled through the driver, stream off the output and capture nodes

Test Application

```
        void outputBufferComplete(FrameBuffer *buffer)
        {
                cout << "Received output buffer" << endl;

                outputFrames_++;

                /* Requeue the buffer for further use. */
                m2mScaler_->output()->queueBuffer(buffer);
        }

        void receiveCaptureBuffer(FrameBuffer *buffer)
        {
                cout << "Received capture buffer" << endl;

                captureFrames_++;
#ifdef DATA_CHECK
                int cookie = buffer->cookie();
                assert(memcmp(outputMappedBuffer_[cookie].planes()[0].data(), bbb_splash_resize_rgb, outputMappedBuffer_[cookie].planes()[0].size()) ==0);
#endif
                /* Requeue the buffer for further use. */
                m2mScaler_->capture()->queueBuffer(buffer);
        }
```

- In the outputBufferComplete callback, simply return the buffer back to driver
- In the recieveCaptureBuffer callback, check to see if the returned buffer's contents match with our expectation

Test Application

```
        void outputBufferComplete(FrameBuffer *buffer)
        {
                cout << "Received output buffer" << endl;

                outputFrames_++;

                /* Requeue the buffer for further use. */
                m2mScaler_->output()->queueBuffer(buffer);
        }

        void receiveCaptureBuffer(FrameBuffer *buffer)
        {
                cout << "Received capture buffer" << endl;

                captureFrames_++;
#ifdef DATA_CHECK
                int cookie = buffer->cookie();
                assert(memcmp(outputMappedBuffer_[cookie].planes()[0].data(), bbb_splash_resize_rgb, outputMappedBuffer_[cookie].planes()[0].size()) ==0);
#endif
                /* Requeue the buffer for further use. */
                m2mScaler_->capture()->queueBuffer(buffer);
        }
```

- In the outputBufferComplete callback, simply return the buffer back to driver
- In the recieveCaptureBuffer callback, check to see if the returned buffer's contents match with out expectation

Build this system

git clone --recurse-submodules -j8 -b elc-2022 https://github.com/karthikpoduval/yoe-distro.git yoe-m2m-elc-2022

cd yoe-m2m-elc-2022

source qemuarm64-envsetup.sh

bitbake v4l2-m2m-example-image

runqemu nographic slirp

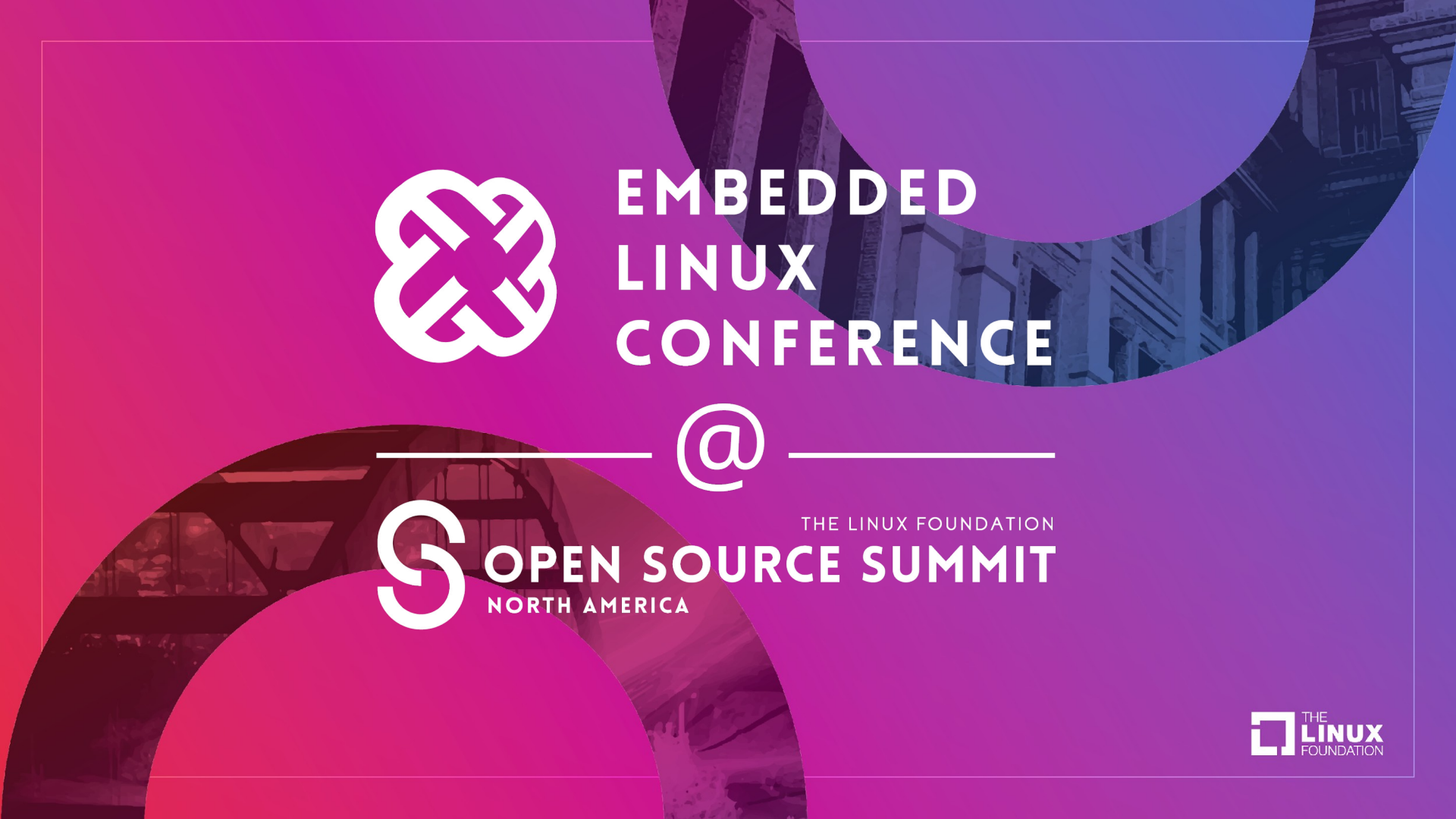
#inside QEMU

m2m-scaler-test

References

- Kernel V4L2 Documentation - https://docs.kernel.org/driver-api/media/v4l2-videobuf2.html
- M2M Scaler Datasheet - https://github.com/karthikpoduval/meta-v4l2-m2m-example/blob/elc-2022/m2m-scaler-datasheet.pdf
- M2M Scaler Driver - https://github.com/karthikpoduval/v4l2-m2m-scaler-driver/blob/elc-2022/v4l2-m2m-scaler.c
- M2M Scaler Test Application - https://github.com/karthikpoduval/meta-v4l2-m2m-example/blob/elc-2022/recipes-multimedia/m2m-scaler-test/src/m2m-example.cpp
- M2M Scaler Device (QEMU) - https://github.com/karthikpoduval/qemu/blob/elc-2022/hw/misc/m2m_scaler.c

Questions ?


EMBEDDED
LINUX
CONFERENCE
@
THE LINUX FOUNDATION
OPEN SOURCE SUMMIT
NORTH AMERICA
THE LINUX FOUNDATION